Chapter 9 Software evolution:

9.1 Evolution processes

9.2 Legacy systems

9.3 Software maintenance

Note: Images excluded due to time constraints

9.1 Evolution processes:

Types of Evolution Processes

- No standard evolution process; varies depending on software type, organizational processes, and team skills.
- Evolution could be formal (e.g., embedded critical systems) or informal (e.g., mobile apps).

Change Proposals

- Driver for system evolution.
- Originates from unimplemented requirements, new requirements, bug reports, or improvement ideas from the development team.
- Cyclical process, continuing throughout system's lifetime.

Change Management Steps

1. **Change Analysis**: Assess which components need to be changed and the impact.
2. **Release Planning**: Decide which proposed changes will be in the next release.
3. **System Implementation**: Implement and validate changes.
4. **System Release**: Release the new version to customers.

Emergency Changes

- Triggered by serious system faults, environment changes, or new legislation.
- Often result in quick, workable solutions at the expense of long-term system structure.

Documentation Concerns

- Risk of code and documentation becoming inconsistent, particularly during emergency changes.
- Minimal documentation is an agile approach to mitigate this issue.

Agile Methods

- Can be used both in development and evolution.
- Seamless transition from development to evolution is possible.
- May need modifications for maintenance and evolution tasks.

Team Handover Challenges

1. **Agile to Plan-based**: Lack of detailed documentation can be problematic.
2. **Plan-based to Agile**: Lack of automated tests and non-refactored code can hinder agile processes.

Agile Techniques

- Test-driven development and automated regression testing are beneficial.
- User stories and Scrum can help prioritize changes.

Overall

- The most appropriate evolution process is context-dependent, and agile methods can adapt to fit maintenance and evolution needs.

9.2 Legacy systems:

9.2.1 Legacy system management:

Overview

- Legacy systems are critical business systems that need to be adapted to changing business practices.
- Organizations must assess their legacy systems to decide the most appropriate evolution strategy.

Strategic Options for Legacy Systems

1. **Scrap the System**: Chosen when the system no longer contributes to business processes.
2. **Leave Unchanged**: Opt for regular maintenance when the system is stable and few changes are requested.
3. **Reengineer**: Improve its maintainability when the system quality is degraded but still relevant.
4. **Replace**: Opt for a new system when the old system cannot continue or off-the-shelf systems can be used.

Business and Technical Perspectives

- Assess legacy systems from both business and technical perspectives.
- Business value and system quality are key metrics to guide the decision-making process.

Factors to Consider for Business Value

1. **Use of the System**: Frequency and user base.
2. **Supported Business Processes**: Obsolescence and inefficiency.
3. **System Dependability**: Affects both technical and business perspectives.
4. **System Outputs**: Importance to business functioning.

Technical Assessment Factors

1. **Environment**: Hardware, support software, supplier stability, and other external factors.
2. **Application Quality**: Number of change requests, user interfaces, and volume of data.

Technical Quality Metrics

1. **Understandability**: Complexity of source code.
2. **Documentation**: Availability and completeness.
3. **Data**: Data model, data duplication, and consistency.
4. **Performance**: Effect on system users.
5. **Programming Language**: Modern compilers and usage.
6. **Configuration Management**: Version management.
7. **Test Data**: Existence and record of regression tests.
8. **Personnel Skills**: Availability of skilled maintainers.

Organizational and Political Considerations

- Decisions are often influenced by organizational and political factors, such as mergers or budget constraints, rather than purely objective assessments.

9.3 Software maintenance:

Overview

- Software maintenance is the act of modifying a system after delivery, often executed by different development groups.
- Changes could be minor, such as correcting coding errors, or major, like adding new functionalities.

Lehman's Laws

- Manny Lehman and Les Belady introduced laws applicable to evolving large-scale software systems, including the notion that programs must continually change to remain useful.

Types of Software Maintenance

1. **Fault Repairs**: Fixing bugs and vulnerabilities.
   - Coding errors are the cheapest to correct, while requirements errors are the most expensive.
2. **Environmental Adaptation**: Adapting the software to new platforms and environments.
   - Necessary when there's a change in the hardware, operating system, or other support software.
3. **Functionality Addition**: Adding new features to support new requirements.

    - Often required in response to organizational or business changes and generally the most extensive type of maintenance.

Maintenance Effort Distribution

- Fault repair constitutes 24%, environmental adaptation 19%, and functionality addition or modification 58% of maintenance effort.

Challenges in Software Maintenance

1. **Team Understanding**: New maintenance teams must spend time understanding the existing system.
2. **Lack of Incentive for Maintainability**: Maintenance and development contracts are often separate, reducing the incentive to write maintainable code.
3. **Unpopularity of Maintenance Work**: Often allocated to less experienced staff and seen as less skilled than development.
4. **Aging Program Structure**: Programs become harder to change as they age.

Solutions and Good Practices

1. **Software Reengineering Techniques**: Used to improve the system structure and understandability.
2. **Architectural Transformations**: Adapt the system to new hardware.
3. **Refactoring**: Improve the quality of the system code.

Business Reluctance

1. **Short-term Planning**: Managers aim to reduce short-term costs, neglecting long-term gains from maintenance.
2. **Development-Maintenance Gap**: Developers are not usually responsible for maintenance, reducing their incentive to make the system maintainable.

Ideal Scenario

- Integrating development and maintenance can make the original development team responsible for the software throughout its lifetime, though this is rare for custom software.

9.3.1 Maintenance prediction:

Overview

- Maintenance prediction aims to assess future changes in a software system and identify the most costly components to change.
- Helps in setting budgets for maintenance and making systems more adaptable.

Factors in Maintenance Prediction

1. **System Interfaces**: The number and complexity of system interfaces influence the likelihood of required changes.
2. **Volatile System Requirements**: Requirements based on organizational policies are more likely to change than those based on stable domain characteristics.

3. **Business Processes**: As these evolve, they generate system change requests, increasing the demand for changes.

Relationship Between Complexity and Maintainability

- Studies have shown that more complex systems or components are more expensive to maintain.
- Complexity measurements can help identify which components should be simplified to reduce maintenance costs.

Process Metrics for Predicting Maintainability

1. **Number of Requests for Corrective Maintenance**: An increasing number may indicate declining maintainability.
2. **Average Time Required for Impact Analysis**: An increase implies more components are affected, lowering maintainability.
3. **Average Time to Implement a Change Request**: Increasing time can indicate declining maintainability.
4. **Number of Outstanding Change Requests**: An increase over time may imply a decline in maintainability.

Cost Estimation Models

- Managers often use a combination of metrics, intuition, and experience to estimate maintenance costs.
- COCOMO 2 model suggests estimates can be based on the effort to understand existing code and the effort to develop new code.

9.3.2 Software reengineering:

Overview

- Software reengineering improves the structure and understandability of legacy systems without altering their functionality.
- It is an alternative to system replacement, offering benefits like reduced risk and cost.

Advantages of Reengineering Over Replacement

1. **Reduced Risk**: Redeveloping business-critical software comes with high risk, including specification errors and development delays.
2. **Reduced Cost**: Reengineering can be significantly cheaper than developing new software.

General Model of Reengineering Process

The reengineering process involves several activities, each aimed at improving different aspects of the system:

1. **Source Code Translation**: Convert the program from an old language to a modern one using translation tools.
2. **Reverse Engineering**: Automatically analyze the program to document its organization and functionality.
3. **Program Structure Improvement**: Analyze and modify the program's control structure for better readability and understandability.

4. **Program Modularization**: Manually group related parts together, remove redundancy, and possibly refactor the architecture.
5. **Data Reengineering**: Modify data to reflect program changes, which might involve redefining database schemas and cleaning up data.

Limitations of Reengineering

- There are practical limits to the extent of improvements achievable through reengineering.
- Major architectural changes or radical data reorganization are very expensive.
- Reengineered systems may not be as maintainable as those built using modern methods.

Cost Spectrum

- Costs of reengineering depend on the extent of work involved.
- Source code translation is usually the cheapest option, while architectural migration is the most expensive.

Interoperability with New Software

- To make the reengineered system work with new software, adaptor services may be required. These present new, better-structured interfaces that can be used by other components.

9.3.3 Refactoring:

Overview

- Refactoring aims to improve the structure and understandability of a program without adding new functionality.
- It serves as "preventative maintenance" that mitigates future problems and is integral to agile development methods.

Refactoring vs. Reengineering

- Unlike reengineering, which is done after a system has been maintained for some time, refactoring is a continuous process throughout the development and evolution of a software system.
- Refactoring aims to prevent code degradation, making future maintenance easier and less costly.

Indicators for Refactoring ("Bad Smells")

1. **Duplicate Code**: Identical or very similar code that appears in multiple places should be refactored into a single method or function.
2. **Long Methods**: Overly long methods should be broken down into shorter, more manageable methods.
3. **Switch (Case) Statements**: These often involve duplication and can usually be refactored using polymorphism in object-oriented languages.
4. **Data Clumping**: Repeated groups of data items should be encapsulated into an object.
5. **Speculative Generality**: Unneeded generality, added for potential future use, can often simply be removed.

Primitive Refactoring Transformations

- Examples include 'Extract Method', 'Consolidate Conditional Expression', and 'Pull up Method'.
- These transformations can be applied individually or in combination to deal with the identified "bad smells."

Tools and Environments

- Interactive development environments like Eclipse often include refactoring support, making it easier to implement these changes.

Limitations

- If a program's structure is significantly degraded, refactoring alone may not suffice, and more expensive and complex design refactoring may be required.

Importance in Long-term Maintenance

- Regular refactoring can significantly reduce the long-term maintenance costs of a program.

Summary

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation.
- Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.
- It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.
- The business value of a legacy system and the quality of the application software and its environment should be assessed to determine whether a system should be replaced, transformed, or maintained.
- There are three types of software maintenance, namely, bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- Software reengineering is concerned with restructuring and redocumenting software to make it easier to understand and change.
- Refactoring, making small program changes that preserve functionality, can be thought of as preventative maintenance.